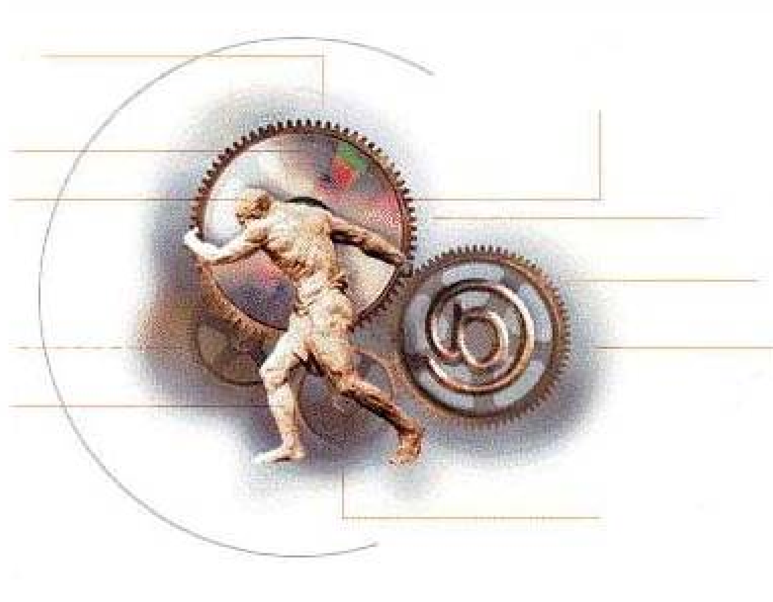
資料結構(Data Structures)

Course 7: Graph



授課教師:陳士杰

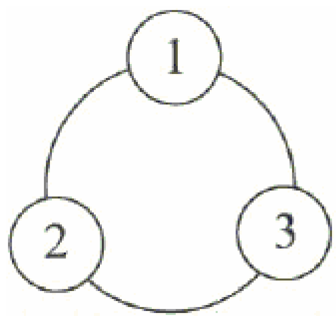
國立聯合大學 資訊管理學系

Outlines

- 本章重點
  - Graph的定義與種類
  - Graph的表示方式
    - Adjacency Matrix (相鄰矩陣)
    - Adjacency List (相鄰串列)
  - DFS與BFS順序
  - AOV Networks與Toplogical
  - AOE Networks

Graph的定義與種類

- Def:
  - 一個圖形(Graph) 是由頂點集合 V 與邊集合 E 所組成，表示如右：G(V, E)。



$G(V, E)$

$V = \{1, 2, 3\}$

$E = \{(1, 2), (1, 3), (2, 3)\}$

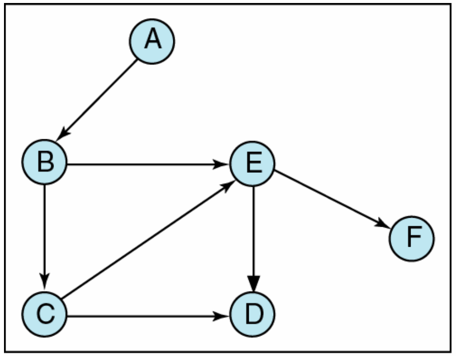
圖形種類:

有向圖 (Directed graph)

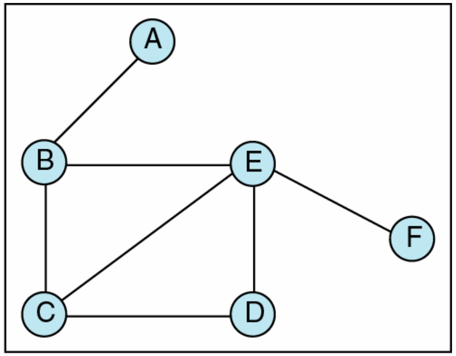
- G = (V, E), 其中V為頂點集合, E為邊集合。邊集合中的每一個邊都有方向性, 以指向下一個頂點。
- (vi, vj) ≠ (vj, vi)。
- 有向圖的邊有時亦稱為弧 (Arc)。

無向圖 (Undirected graph)

- G = (V, E), 其中V為頂點集合, E為邊集合。邊集合中的每一個邊都沒有方向性。
- (vi, vj) = (vj, vi) 。

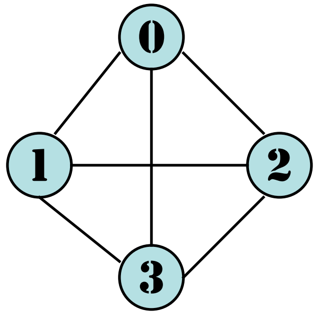


(a) Directed graph



(b) Undirected graph

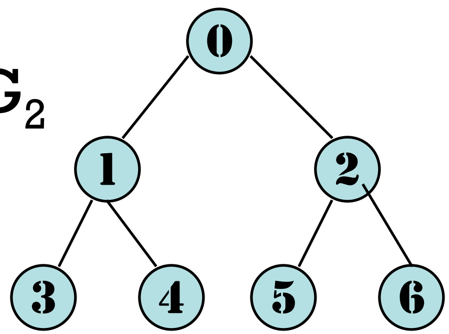
G_1



$V(G_1)=\{0,1,2,3\}$

$E(G_1)=\{(0,1), (0,2),(0,3),(1,2),(1,3),(2,3)\}$

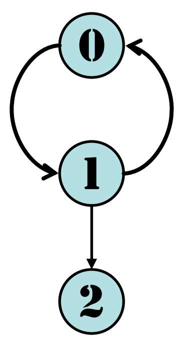
G_2



$V(G_2)=\{0,1,2,3,4,5,6\}$

$E(G_2)=\{(0,1), (0,2),(1,3),(1,4),(2,5),(2,6)\}$

G_3

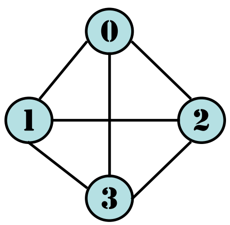


$V(G_3)=\{0,1,2\}$

$E(G_3)=\{(0,1), (1,0),(1,2)\}$

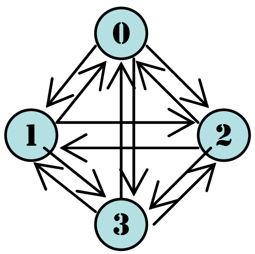
Complete Graph (完整圖):

- 一個完整圖(complete graph)是一個擁有最多非重複邊線的圖
  - **無向圖**:若圖具有n個頂點，則具有最多的非重複邊個數達 n(n-1)/2 時，此圖稱為完整圖。
    - 例:



n(n-1)/2 = 4(4-1)/2 = 6 (非重複邊個數)

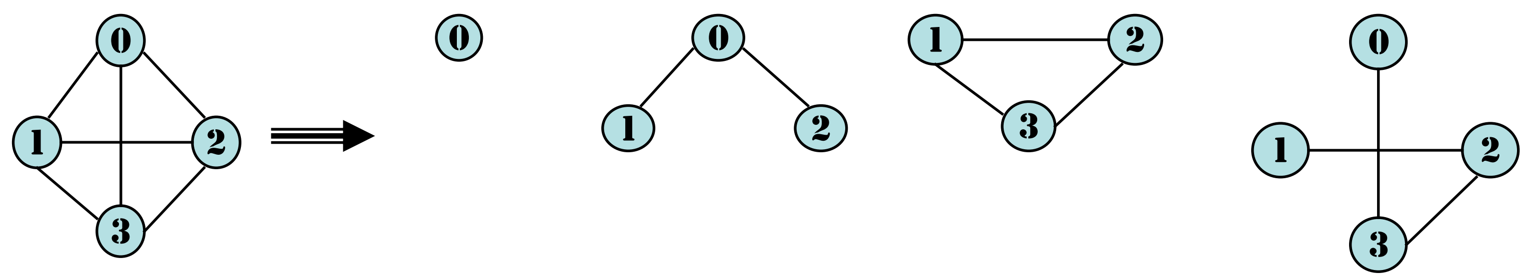
  - **有向圖**:若圖具有n個頂點，則具有最多的非重複邊個數達 n(n-1) 時，此圖稱為完整圖。
    - 例:

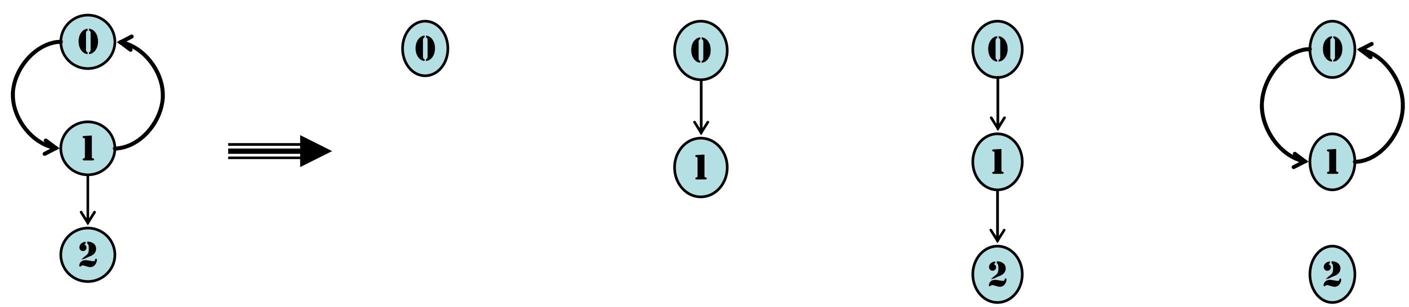


n(n-1) = 4(4-1) = 12 (非重複邊個數)

Subgraph (子圖):

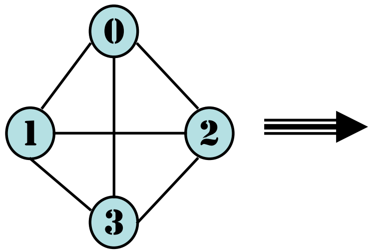
- 若 G'是圖形G的子圖, 則表示 V(G')$\subseteq$ V(G) 且 E(G') $\subseteq$ E(G)。
- 例: (以下子圖未完全列出)



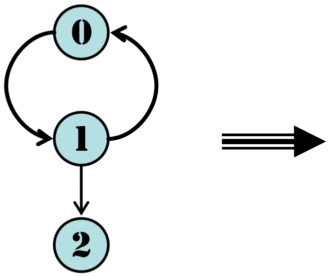


Path (路徑):

- 在圖形G中，由頂點 v_p 到 v_q 的路徑是一連串頂點的集合。
- 例：(以下路徑未完全列出)



- (0, 1)、(1, 3)、(3, 2) 或 (0, 1, 3, 2)
- (1, 2)、(2, 3) 或 (1, 2, 3)
- (0, 1)、(1, 3)、(3, 1) 或 (0, 1, 3, 1)



- (0, 1)、(1, 2) 或 (0, 1, 2)
- (1, 0)、(0, 1)、(1, 2) 或 (1, 0, 1, 2)

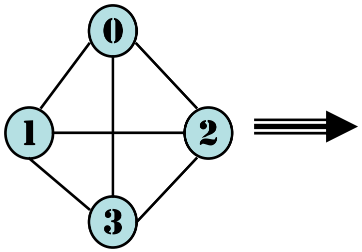
Path Length (路徑長度):

- 路徑上所包含的邊之數目。

Simple Path (簡單路徑):

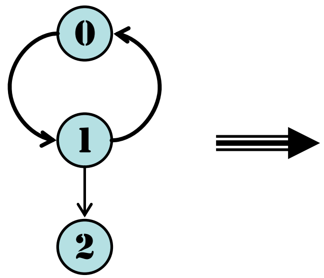
- 路徑上除了起點和終點可以相同之外，其餘頂點均不相同。

例:



- (0, 1, 3, 2)
- (1, 2, 3)
- (0, 1, 3, 1)

長度?	簡單路徑?
3	Yes
2	Yes
3	No

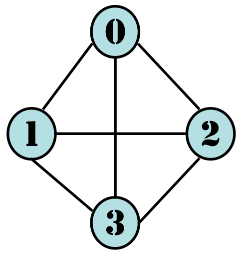


- (0, 1, 2)
- (1, 0, 1, 2)

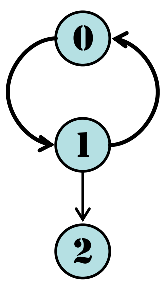
長度?	簡單路徑?
2	Yes
3	No

Cycle (迴圈)：

- 是一個簡單路徑，且起點與終點相同。
- 例：



- (0, 1, 3, 0)
- (1, 2, 3, 1)
- …

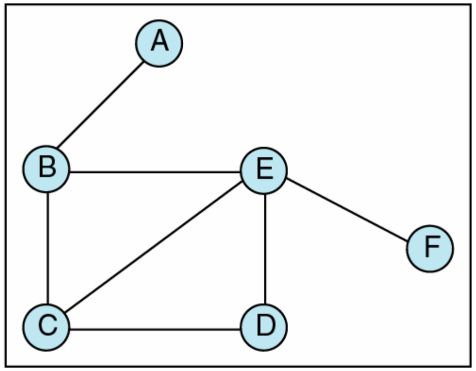


- (1, 0, 1)

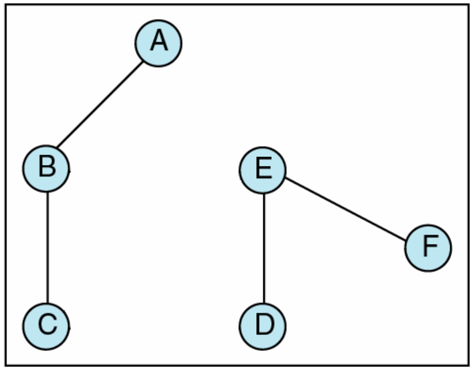
Connected (連通) [對無向圖而言]

- Def: 在一undirected graph中，任何成對頂點之間皆有路徑存在。

Connected



Unconnected



Connected [對有向圖而言]

Strongly connected (強連通)

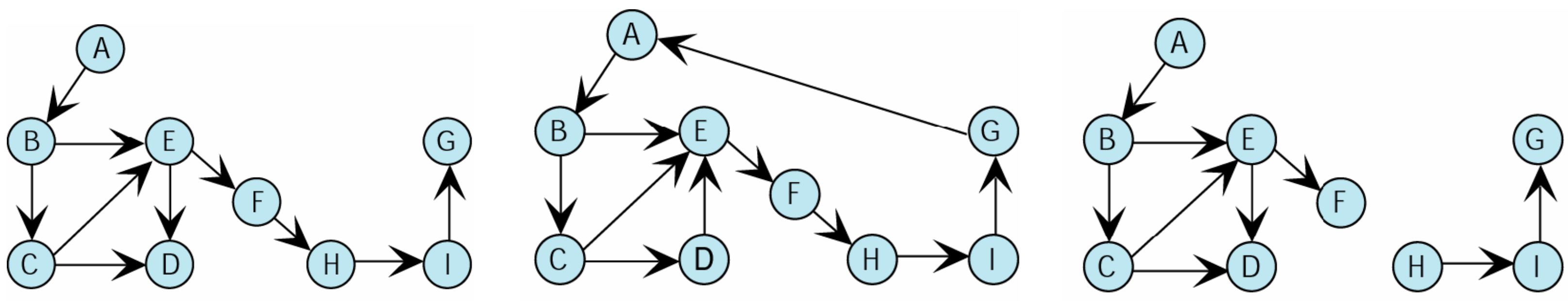
- 在有向圖形中，任何成對頂點之間皆有路徑可以相互到達對方。

Weakly connected (弱連通)

- 在有向圖中，至少有兩個頂點無法以有向路徑相互到達對方。
- 即：僅某一頂點可到達另一點，而另一點卻無法走回到對方。

Disjoint (不相連)

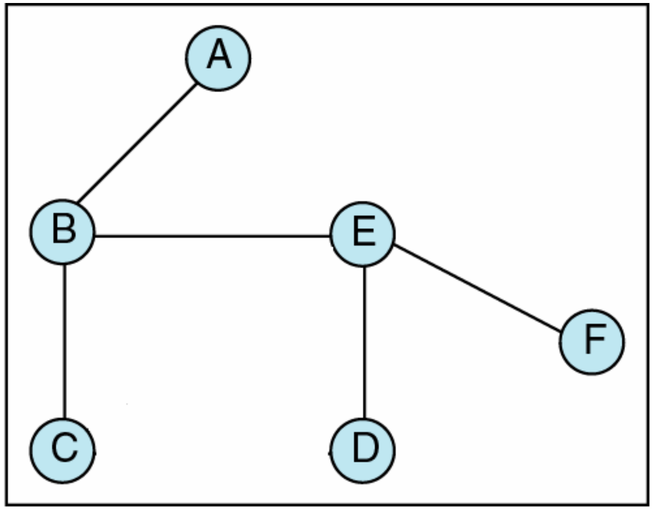
- 一個有向圖形被切割成多個互不相連的子圖。



(a) Weakly connected (b) Strongly connected (c) Disjoint graph

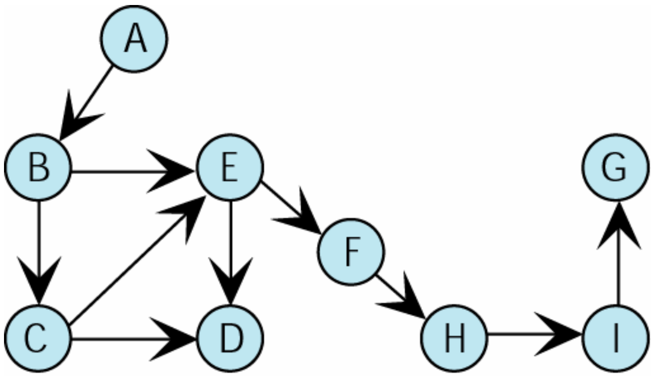
Degree (分枝度):

- (無向圖) 一個頂點的分枝度，為連接至該頂點的<u>邊之個數</u>。
  - 範例:



  - The degrees of the nodes A, C, D, F = 1
  - The degrees of the nodes B, E = 3

- (有向圖) 一個頂點的分枝度，是由該頂點的入分枝度 (**Indegree**) 與出分枝度 (**Outdegree**) 加總而得。
  - **Outdegree**：一個頂點的出分枝度為離開該頂點的邊之個數。.
  - **Indegree**：一個頂點的入分枝度為進入該頂點的邊之個數。.
  - 範例:



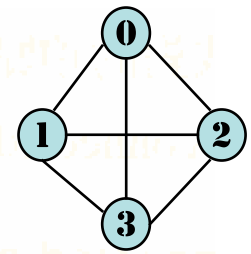
  - B 的 Indegree = 1; B 的 outdegree = 2; B 的 degree = 3
  - E 的 indegree = 2; E 的 outdegree = 2; E 的 degree = 4

Degree與邊的關係:

- (無向圖):

$$\sum_{i=1}^{n} d_i = 2e \text{ ，} \therefore e = \frac{\sum d_i}{2}$$

(d_i：指 V_i的 degree)

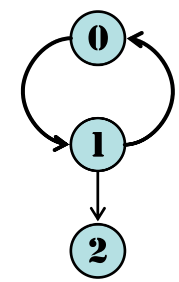


- e = (3+3+3+3) / 2
  = 6

- (有向圖):

$$e = \sum_{i=1}^{n} (\underline{\text{Indegree}_i}) = \sum_{i=1}^{n} (\underline{\text{Outdegree}_i})$$

(指 V_i的 indegree)　　(指 V_i的 outdegree)



- e = 3
(僅看Indegree 或 Outdegree的個數)

圖形的儲存結構 (Graph Storage Structures)

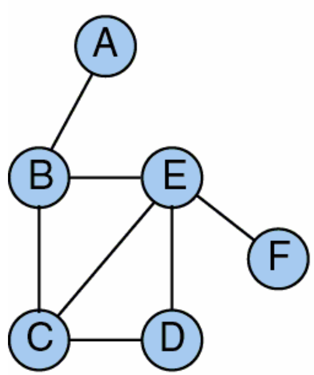
- 圖形在系統中的表示方式
- 若要在系統中表示一個圖形，需要儲存兩個集合：
  - 圖形的頂點 (**Vertices**)
  - 圖形的邊 （**Edges** or **arcs**)
- 常用的表示方式：
  - Adjacency Matrix; 相鄰矩陣
  - Adjacency List; 相鄰串列
  - *Multiple Adjacency List; 相鄰多元串列*
  - *Index Array; 索引陣列*

Adjacency Matrix (相鄰矩陣)

- 對無向圖而言:
  - Def: G=(V, E)為一無向圖, |V| = n。宣告一個n×n的二維矩陣A, 其中:

$$A[i\,,\,j] = \begin{cases} 0, \text{ if } (V_i, V_j) \notin E \\ 1, \text{ if } (V_i, V_j) \in E \end{cases}$$

  - 範例:



	A	B	C	D	E	F
A	0	1	0	0	0	0
B	1	0	1	0	1	0
C	0	1	0	1	1	0
D	0	0	1	0	1	0
E	0	1	1	1	0	1
F	0	0	0	0	1	0

Adjacency matrix

Adjacency matrix for non-directed graph

- 特質：此Matrix必為對稱 (Symmetric)
  - **說明**：∵無向圖中，$(v_i, v_j) = (v_j, v_i)$，∴ A[i, j] = A[j, i]。儲存時可用上三角或下三角陣列。
- 運作：
  - 判斷邊 (v_i, v_j) 是否存在。

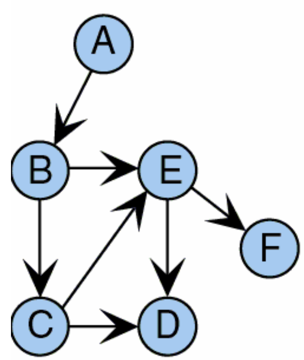
    $$A[i,j]\begin{cases} =1，存在 \\ =0，不存在 \end{cases}$$

    - 作法：檢查A[i, j]的值是否為 1
    - 時間複雜度：O(1)
  - 求頂點 v_i 的分枝度
    - 作法：求第 i 列或第 i 行之元素值總和
    - 時間複雜度：O(n) (∵需用一個迴圈)
  - 求圖形的邊數
    - 作法：$e = \sum_{i=1}^{n} Di \Big/ 2 = \sum_{i=1}^{n} \sum_{j=1}^{n} A[i, j] \Big/ 2$
    - 時間複雜度：$O(n^2)$

- 對有向圖而言:
  - Def: G=(V, E)為一有向圖, |V| = n。宣告一個n×n的二維矩陣A, 其中:

$$A[i\,,j]=\begin{cases}0, \text{ if } (V_i, V_j)\notin E\\ 1, \text{ if } (V_i, V_j)\in E\end{cases}$$

  - 範例: (以Out-degree 為主)



	A	B	C	D	E	F
A	0	1	0	0	0	0
B	0	0	1	0	1	0
C	0	0	0	1	1	0
D	0	0	0	0	0	0
E	0	0	0	1	0	1
F	0	0	0	0	0	0

Adjacency matrix

Adjacency matrix for directed graph

- 特質：此Matrix不一定是對稱 (Symmetric)
  - **說明**：∵有向圖中，(v_i, v_j) 存在，不見得 (v_j, v_i) 也會存在。
- 運作：
  - 判斷邊 (v_i, v_j) 是否存在。
    - 作法：檢查A[i, j]的值是否為 1 (同無向圖)
    - 時間複雜度：O(1)

$$A[i, j]\begin{cases} =1\text{，存在} \\ =0\text{，不存在} \end{cases}$$

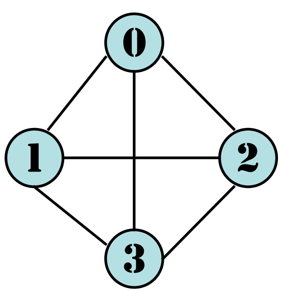
  - 求頂點 v_i 的出分枝度與入分枝度 (若矩陣是以Out-degree為主)
    - 作法：

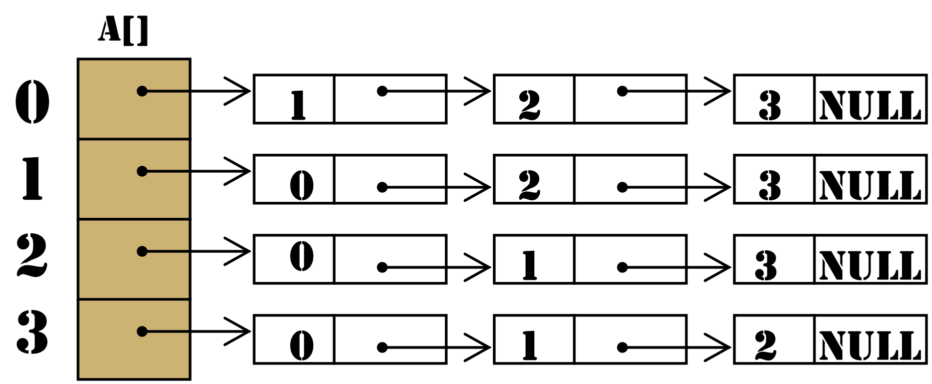
      (1) Out-degree：求第 i 列之元素值總和。

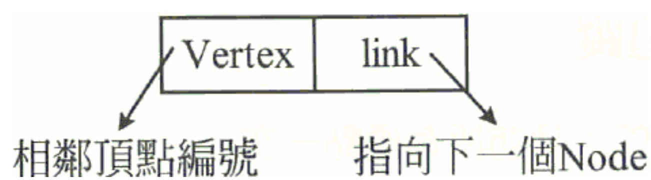
      (2) In-degree：求第 i 行之元素值總和。
    - 時間複雜度：O(n) (∵需用一個迴圈)
  - 求圖形的邊數
    - 作法：$e = \sum_{i=1}^{n} D_i^{Out} = \sum_{i=1}^{n} D_i^{In} = \sum_{i=1}^{n}\sum_{j=1}^{n} A[i, j]$
    - 時間複雜度：$O(n^2)$

Adjacency List (相鄰串列)

- 對無向圖而言:
  - Def: 若 G=(V, E)為一無向圖, |V| = n 且|E|=e, 則需使用 **n 條相鄰串列**與**一個Size為n的一維矩陣A[1, ..., n]**, 其中A[i] 所存放的是頂點 i 的相鄰串列, 此串列中每個Node皆記錄著與頂點 i 相鄰之其它頂點。
  - 範例:







- 特質：在無向圖中，相鄰串列的Node總數 = 2e
  - **說明**：∵無向圖中，$(v_i, v_j) = (v_j, v_i)$，∴在頂點為 i 的相鄰串列中會加入 v_j 的Node；同時，在頂點為 j 的相鄰串列中會加入 v_i 的Node。
- 運作：
  - 判斷邊 (v_i, v_j) 是否存在。
    - 作法：檢查頂點為 i 的相鄰串列中，是否有Node j 存在。(Yes: 存在, No: 不存在)
    - 時間複雜度：O(e) (∵與 v_i 相連的邊之個數有關，∴比相鄰矩陣花時間)
  - 求頂點 v_i 的分枝度
    - 作法：求頂點 i 之相鄰串列的長度 (即：節點總數)
    - 時間複雜度：O(e)

求圖形的邊數

- 作法:

$$e = \sum_{i=1}^{n} Di \Big/ 2 = \sum_{i=1}^{n} (\text{頂點 } i \text{ 之相鄰串列長度}) \Big/ 2$$

$$= \frac{(\text{所有串列之節點總數})}{2}$$

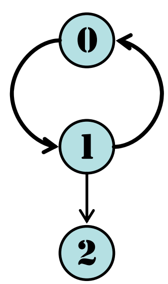
- 時間複雜度: O(n + e)

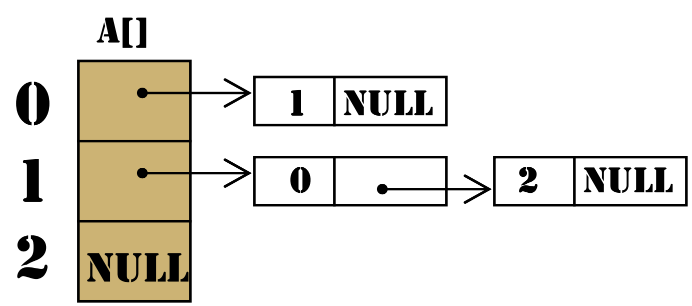
時間複雜度 O(n+e) 的進一步分析:

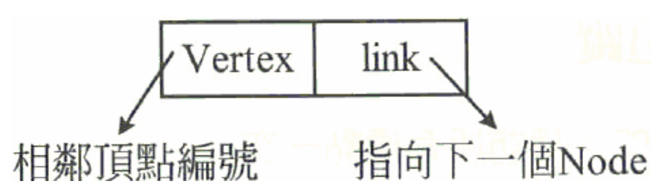
- **當圖形的邊很少的情況**: 一個無向圖可以保持連通的最少邊數為 e = n−1。因此, O(n+e) = O(2n−1) ≒O(n)
- **當圖形的邊很多的情況**: 一個擁有最多邊數的圖形是完整圖 (Complete Graph), 即: e = n(n−1)/2。因此, O(n+e) ≒ O(n^2)
- 由上述兩種情況分析, 當圖形的邊很多的情況, 使用相鄰串列來求圖形的邊數不見得比相鄰矩陣有利。然而, 時間複雜度雖然可能會與相鄰矩陣相同, 但是相鄰串列在資料結構上多存放了link, 需要更多的記憶體空間來執行。

對有向圖而言：

- Def: 若 G=(V, E)為一有向圖，|V| = n 且|E|=e，則需使用 n 條相鄰串列與一個Size為n的一維矩陣A[1, …, n]，其中A[i] 所存放的是頂點 i 的相鄰串列，此串列中每個Node皆記錄著與頂點 i 相鄰之其它頂點。
- 範例：(以Out-degree為主)







- 特質：在有向圖中，相鄰串列的Node總數 = e
  - **說明**：∵有向圖中，(v_i, v_j) 存在，不見得 (v_j, v_i) 也會存在。
- 運作：
  - 判斷邊 (v_i, v_j) 是否存在。
    - 作法：檢查頂點為 i 的相鄰串列中，是否有Node j 存在。(Yes: 存在, No: 不存在)
    - 時間複雜度：O(e) (∵與 v_i 相連的邊之個數有關)
  - 求頂點 v_i 的分枝度
    - Out-degree：求頂點 i 之相鄰串列的長度 (即：節點總數；時間複雜度：O(e))
    - In-degree：檢查所有串列，統計Node i 出現的次數 (時間複雜度：O(n+e))

改進：採用 "反轉相鄰串列 (Inverse Adjacency List)"
- 以 In-degree角度建立list，故在求In-degree時較方便，僅需O(e)。
- 但在求Out-degree時麻煩，依舊要O(n+e)。⇔ 問題根本沒解決!!

求圖形的邊數

- 作法:

$$e = \sum_{i=1}^{n} Di = \sum_{i=1}^{n} (\text{頂點 } i \text{ 之相鄰串列長度})$$
$$= (\text{所有串列之節點總數})$$

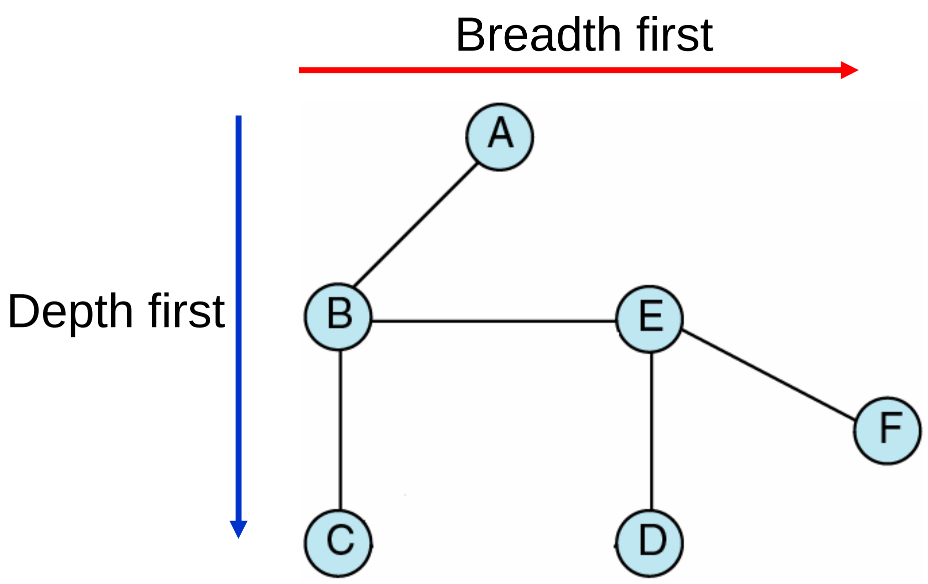
- 時間複雜度:O(n + e)

相鄰矩陣 v.s. 相鄰串列

	相鄰矩陣	相鄰串列	
優點	判斷邊是否存在較容易 (∵O(1))	判斷邊是否存在較麻煩 (∵O(e))	缺點
優點	若為Complete Graph，較節省空間	若為Complete Graph，較浪費空間 (∵需額外的link空間)	缺點
缺點	當頂點個數多，而邊數極少時，會形成Sparse Matrix，較浪費空間	當頂點個數多，而邊數極少時，會形成Sparse Matrix，較相鄰矩陣節省空間	優點
缺點	對某些運作較為麻煩，如：求算邊數、判斷是否連通…。	對某些運作較相鄰矩陣簡單，如：求算邊數、判斷是否連通…。(除非 e 很大)	優點

圖形追蹤 (Traverse Graph)

- 目的:
  - 圖形中的所有頂點都被拜訪到，且僅被拜訪到一次。
- 拜訪方式有兩種:
  - **Depth First Search (DFS;深先搜尋，需利用Stack)**
  - **Breadth First Search (BFS;廣先搜尋，需利用Queue)**
- 應用:
  - 檢查Graph是否連通？
  - 找出此圖的連通單元
  - …

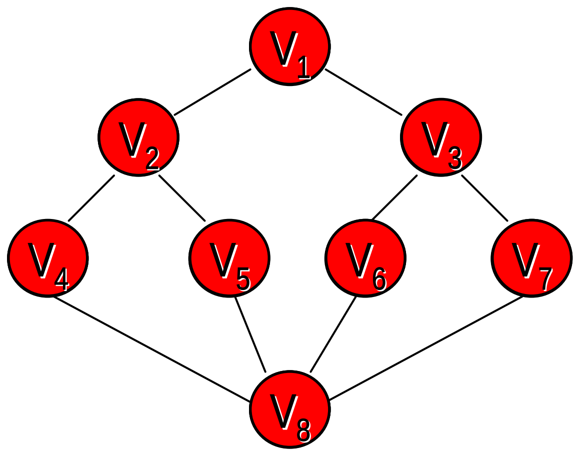


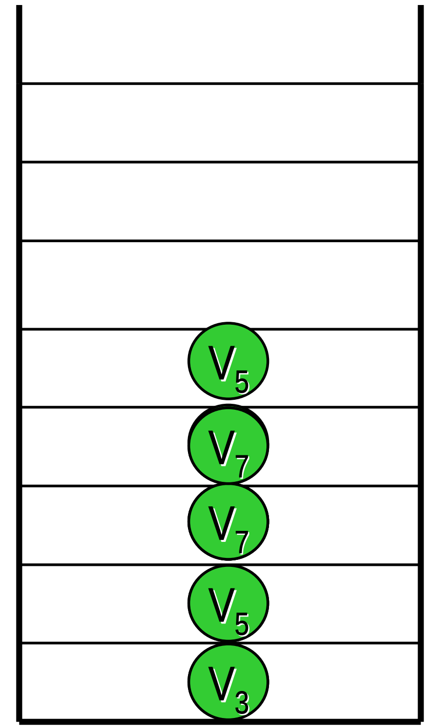
- 不論是採用何種圖形追蹤方法，在實作上皆可引入一個 **visited flag** (拜訪旗標)，以指出頂點的目前狀況。
  - Flag = 0：尚未拜訪 (not processed)
  - Flag = 1：拜訪中 (in queue or stack)
  - Flag = 2：已拜訪 (processed)

Depth-First Traversal

- 拜訪過程：
  - 走訪起始頂點 v，然後選擇一個相鄰至v且尚未被拜訪過的頂點w；以w為起始點再做Depth-First追蹤。如果從任何已拜訪過的頂點，都無法再拜訪到一個尚未被走過的頂點時，則結束拜訪。
- 包含**遞迴**應用的概念，因此可利用**Stack**保存走訪過程中間所走過的點。
- Steps:
  - 選擇一起始拜訪頂點 (可任選)，將它Push到stack中。
  - 若Stack不為空，則
    - 從Stack中Pop一個頂點 (視為**已拜訪頂點**)，並將此頂點所有相鄰之其它未拜訪頂點Push到Stack中。(***重覆執行***)
    - 若所有頂點皆已被拜訪過，而Stack仍不為空時，則將Stack清空
  - 若Stack為空，則追蹤程序完成。

- 例:





Vertex	V_1	V_2	V_3	V_4	V_5	V_6	V_7	V_8
Flag	2	2	2	2	2	2	2	2

Ans:

- Note:
  - DFS之順序**並不唯一**
  - 除非規定 "拜訪時, 依Node編號**由小到大**拜訪" 才會唯一。

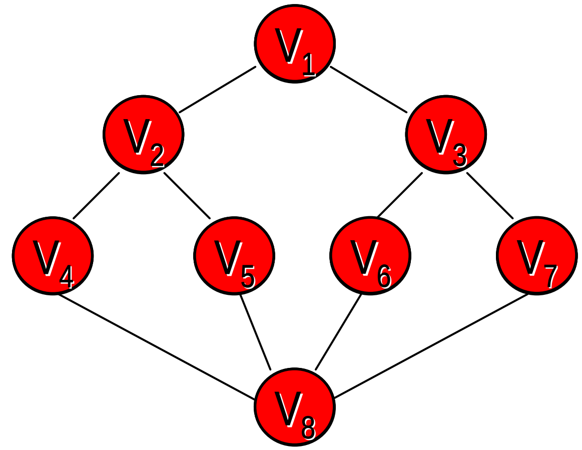
※範例練習※

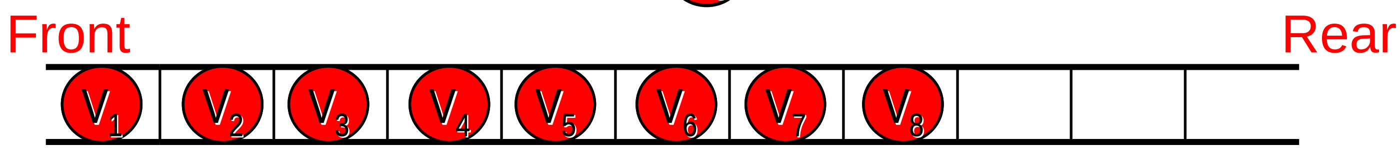
- 見上圖，下列何者不為DFS之Order?
  - a. 1, 2, 5, 8, 6, 3, 7, 4
  - b. 1, 3, 6, 8, 5, 2, 4, 7
  - c. 1, 3, 7, 8, 6, 5, 2, 4
  - d. 1, 2, 4, 8, 6, 3, 7, 5
  - e. 1, 3, 7, 8, 5, 4, 2, 6

Ans: e

- 拜訪過程：
  - 由起始頂點 v 開始走訪。所有相鄰至v且尚未被拜訪過的頂點，都會在下個步驟裡一一被走訪。而相鄰至這些被走訪頂點且尚未走過的頂點，又將被一一走訪；重複上述，直到無頂點可被拜訪為止。
- Steps:
  - 選擇一起始拜訪頂點 (可任選)，將它放入Queue中。
  - 若Queue不為空，則
    - 從Queue的前端移出一個頂點 (視為**已拜訪頂點**)，並將此頂點所有相鄰之其它未拜訪頂點放入Queue中。(重覆執行)
  - 若Queue為空，則追蹤程序完成。

- 例:



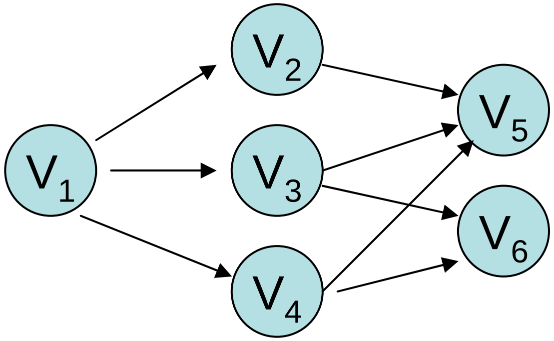


Vertex	V_1	V_2	V_3	V_4	V_5	V_6	V_7	V_8
Flag	2	2	2	2	2	2	2	2

Ans:

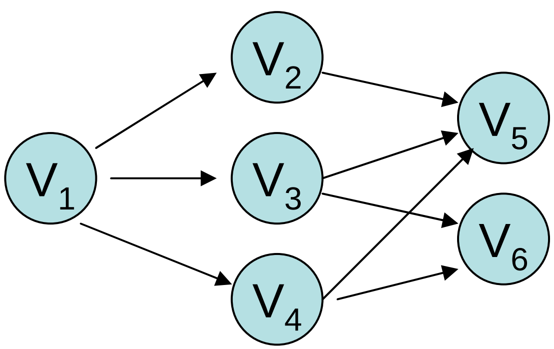
- Note:
  - BFS之順序**並不唯一**
  - 除非規定 "拜訪時，依Node編號**由小到大**拜訪" 才會唯一。

- AOV (Activity On Vertex) Network:
  - Def: 假設G = <V, E>為一個Directed grapth, 其中**Vertex**表示工作(Activity), **Edge**表示工作執行之先後次序關係。
  - E.g. $V_i \rightarrow V_j$, 表示V_i工作必須先於V_j執行, 稱G為AOV Network.
  - 例:



  - 應用: 求合理的工作執行順序 ⇨ 即:Topological Order (拓樸順序)

- Def: 給定一個**不具Cycle**的AOV Network，則可以定出 **≥1組**Vertex (或稱job) 的拜訪順序，且此順序須滿足: “若在AOV Network，V_i為V_j的前導，則在此順序中，V_i必定出現在V_j之前。”
- 例: 請列出下圖的其中一組Topological Order。



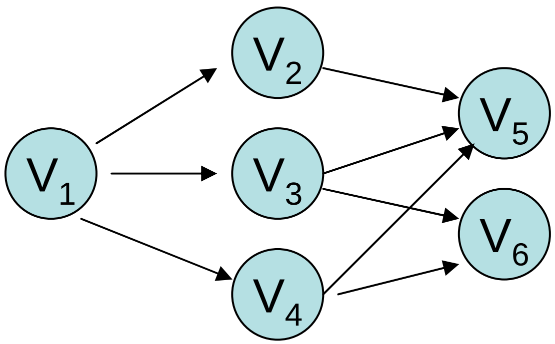
Sol: V_1, V_2, V_3, V_4, V_5, V_6

① 找出一個**無前導**的頂點 (Indegree = 0)

② 將此頂點輸出，且**刪除此點所Leading-out之edge**.

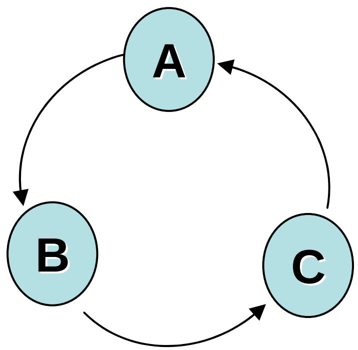
③ Repeat ①~②，直到**所有Vertex已輸出**，或剩下的點**皆有前導存在**

④ If “不是所有點皆輸出” then “No Topological Order存在”



- Sol:

- Note:
  - 若AOV Network有cycle，則無Topological Order。(∵無法決定誰先做!!)
  - 例：



  - 不具cycle的AOV Network，其Topological Order ≥ 1組。(不一定唯一)

AOV Network之資料結構表示

- 作法:
  - 利用 Adjacency List, 並在相鄰串列Vertex[1…n]中多加一個欄位:count欄, 用以記錄vertex的射入邊數 (即:in-degree)。
  - 如何實作 "**刪除某vertex所Leading-out之edge**"的動作？
    - 從該vertex的相鄰串列中, 找出與之相鄰的其它頂點
    - 將這些頂點的count欄之值**減1**
      - ∵ 刪除某vertex所Leading-out之edge = 降低與該vertex相鄰之所有頂點的in-degree數目。(沒有實質的 "**刪除邊**" 之動作)

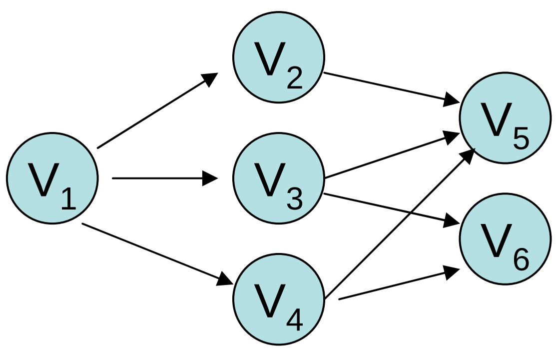
範例

① 找出一個**無前導**的頂點 (Indegree = 0)

② 將此頂點輸出，且**刪除此點所Leading-out之edge**.

③ Repeat ①~②，直到**所有Vertex已輸出**，或剩下的點**皆有前導存在**

④ If "不是所有點皆輸出" then "No Topological Order存在"



Vertex

	Count	Pointer
V_1	0	→ V_2 → V_3 → V_4 → nil
V_2	0	→ V_5 → nil
V_3	0	→ V_5 → V_6 → nil
V_4	0	→ V_5 → V_6 → nil
V_5	0	→ nil
V_6	0	→ nil

- Sol:

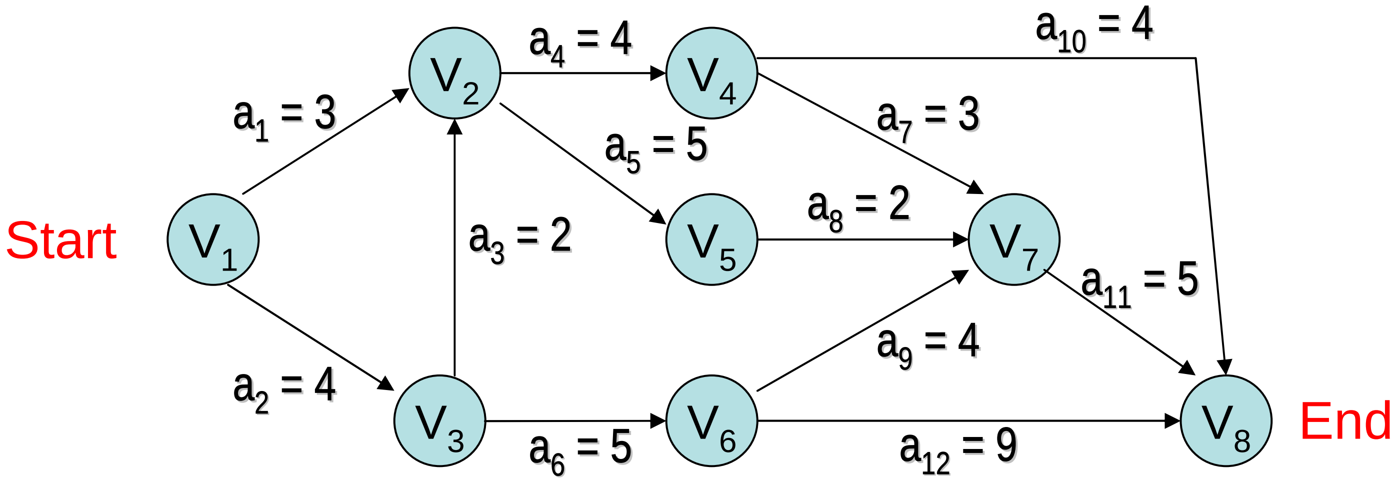
補　充

Graph其它常見議題

- Spanning Tree (擴張樹)
- Min. Spanning Tree (最小擴張樹; 最小成本擴張樹)
- Shortest Path Problem (最短路徑問題)
  - Single Source to Other Destinations (**單一頂點到其它頂點**的最短路徑)
  - All Pairs of Vertex (**所有頂點對之間**的最短路徑)

(於演算法課程講授，請見演算法數位課程)

- AOE (Activity On Edge) Network:
  - Def: G = (V, E)有向圖，以**Edge表示工作 (Activity)**，**Vertex表示事件 (Event)**，Edge上具有加權值，表示工作完成所需的時數。
  - Note:
    - 事件發生，射出工作 (leading out) 才可以開始執行
    - 所有射入工作完成，事件才會發生
  - 例：



1. 完成此計畫所需之最少(最起碼)的工作時數?

- 求從Start →End事件之**最長路徑**長度
- 即: 求**Critical path (臨界路徑)**的長度

- 上圖的可能路徑有(取部份為例):
  - $V_1 \rightarrow V_3 \rightarrow V_6 \rightarrow V_8$ (時數= 18)
  - $V_1 \rightarrow V_3 \rightarrow V_2 \rightarrow V_5 \rightarrow V_7 \rightarrow V_8$ (時數= 18)
  - $V_1 \rightarrow V_3 \rightarrow V_6 \rightarrow V_7 \rightarrow V_8$ (時數= 18)
  - $V_1 \rightarrow V_3 \rightarrow V_2 \rightarrow V_4 \rightarrow V_7 \rightarrow V_8$ (時數= 18)
  - $V_1 \rightarrow V_3 \rightarrow V_2 \rightarrow V_4 \rightarrow V_8$ (時數= 14)
  - $V_1 \rightarrow V_2 \rightarrow V_4 \rightarrow V_8$ (時數= 11)
  - 最長路徑的長度為18, 因此有四條路徑為Critical path.

2. 求Critical Task (臨界任務)

- 加速此工作可以**加速計畫之完成** (縮短計畫完成時數)
- 即: 求**所有Critical Path的交集工作**

- 前例所有的臨界路徑有:
  - $V_1 \rightarrow V_3 \rightarrow V_6 \rightarrow V_8$ (時數= 18)
  - $V_1 \rightarrow V_3 \rightarrow V_2 \rightarrow V_5 \rightarrow V_7 \rightarrow V_8$ (時數= 18)
  - $V_1 \rightarrow V_3 \rightarrow V_6 \rightarrow V_7 \rightarrow V_8$ (時數= 18)
  - $V_1 \rightarrow V_3 \rightarrow V_2 \rightarrow V_4 \rightarrow V_7 \rightarrow V_8$ (時數= 18)
  - 交集的工作為a_2，此即為Critical Task。若加速此工作，則有助於縮短完成時間。